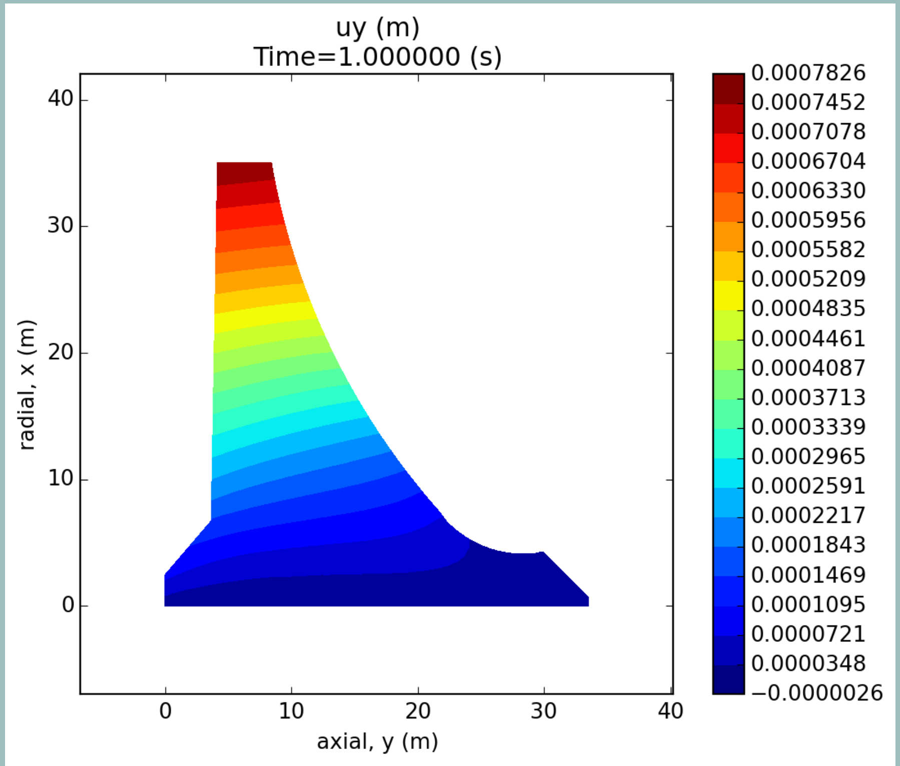
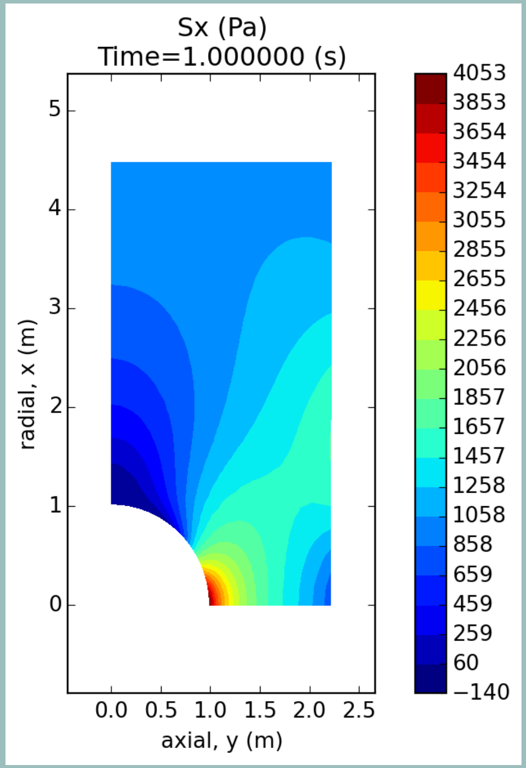
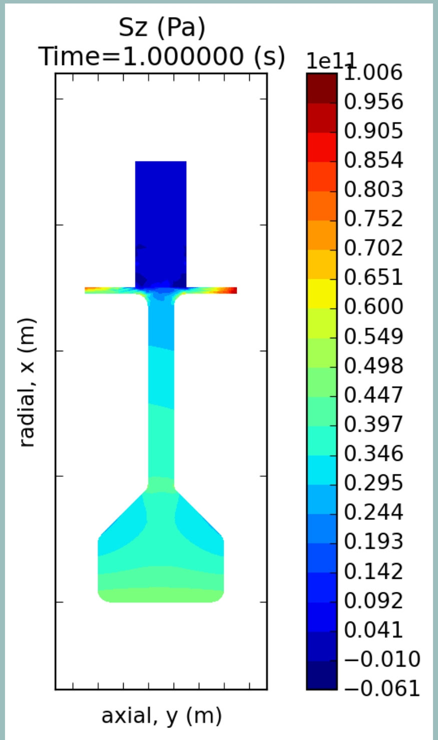






WRAPPING EXISTING TOOLS IN PYTHON: PYCALCULIX

[HTTPS://GITHUB.COM/SPACETHER/PYCALCULIX](https://github.com/spacether/pycalculix)

Presentation to Boston Python User Group
2014-12-11
Justin Black
Mechanical Engineer
Justin.a.black@gmail.com
www.justinablack.com

WHAT AND WHY

What:

Python API to build, solve and analyze mechanical engineering Finite Element Analysis (FEA) models of parts

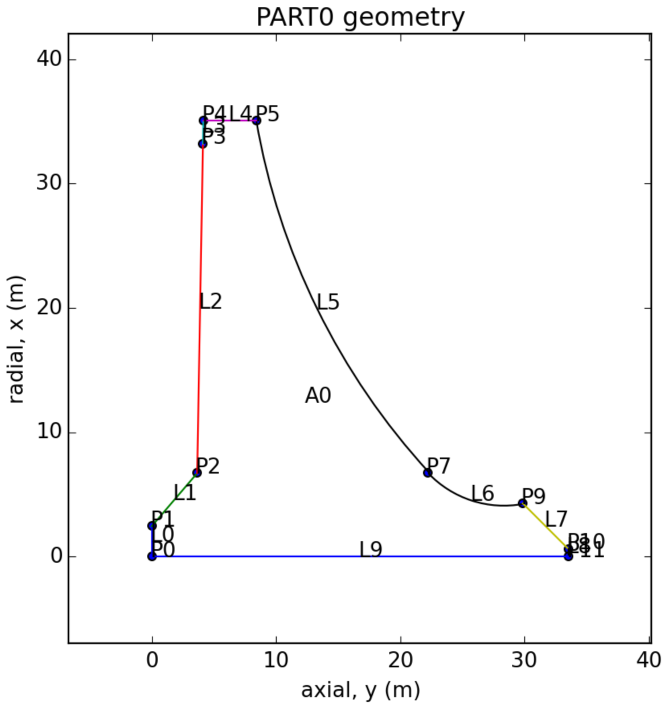
Forces, displacements, gravity etc can be applied to a part and displacements and stresses can be displayed and queried

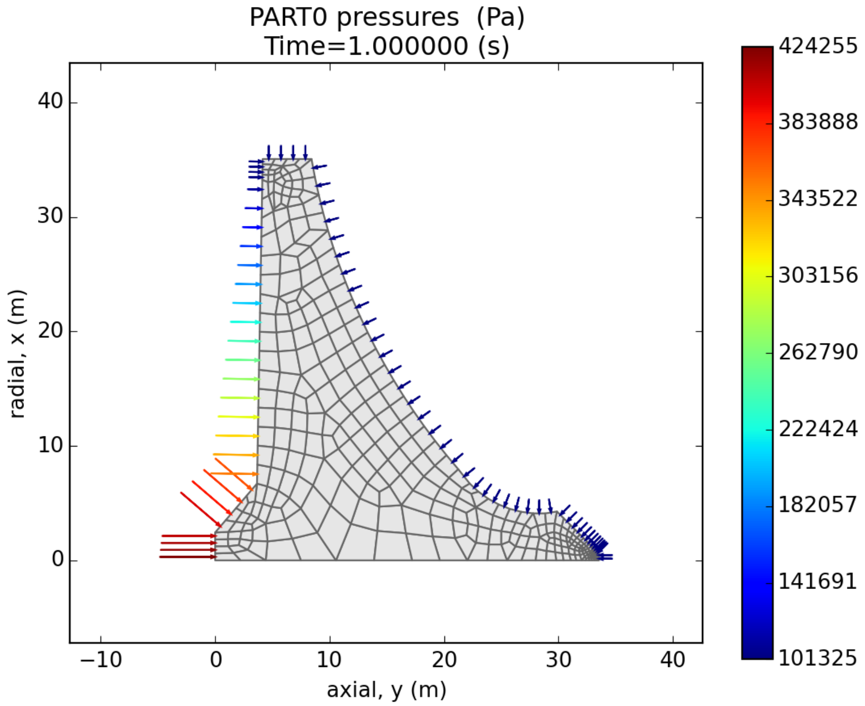
Why?

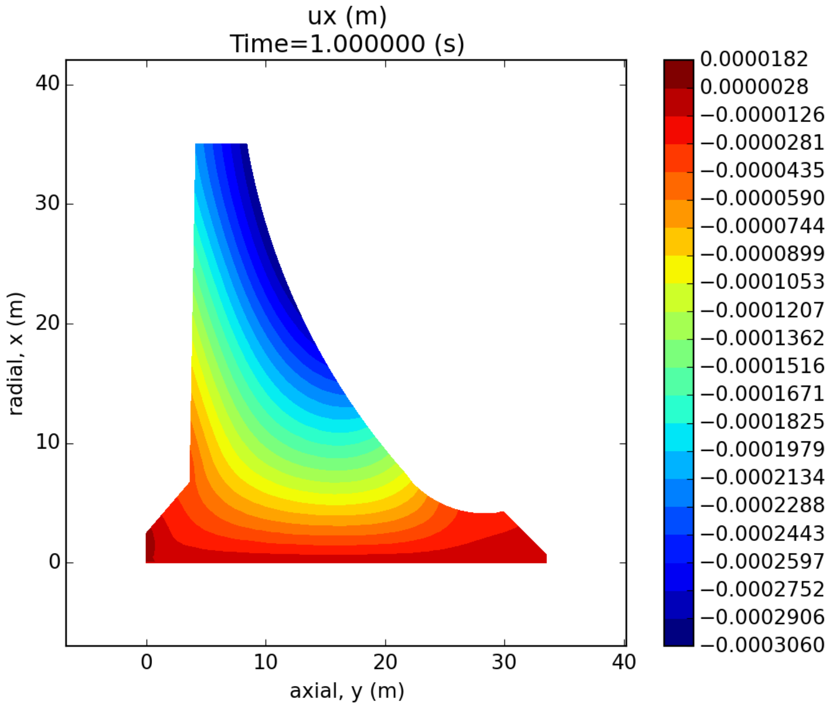
Existing free tools are very capable but not very automatable or user friendly

With current workflow you have to learn 4 programs, I reduced that to one

EXAMPLE: ANALYZING A CONCRETE DAM







Make part
Assign material

Mesh part
Apply constraints
Apply pressures
Apply gravity

Solve model
View Results

PROGRAM REQUIREMENTS + TACTICS

When I started I wrote pseudocode of how I wanted to call my library

Below are lower level requirements

GEOMETRY: store points, lines, areas, groups of areas

MESH: export geometry file to be meshed and import results of meshing

MODEL: apply loads + constraints to geometry

SOLVER: write solver input file

RESULTS: import and query results

KEY CHALLENGES

1. All geometry items must have unique ids and have child nodes elements, faces
2. How do we plot just our part and not an interpolation blob?
3. Results file format must be brought back into python for querying and plotting

UNIQUELY IDENTIFY GEOMETRY, PG1

Geometry is made of primitives:

Part → Areas → Lines → Points

Each primitive must have a unique identity so the mesher knows which lines make an area.

I chose to number each primitive: 0,1,2,3 etc

A dict sounds like a great way to store them, but what if we need to renumber them later. What if we delete item #1?

Instead, I opted for lists of items, where each item has an id number

UNIQUELY IDENTIFY GEOMETRY, PG2

```python
class FeaModel():
    def __init__(s, fname):
        s.fname = fname
        s.points = Item_List()
        s.lines = Item_List()
        s.areas = Item_List()
        s.parts = Item_List()
```

```python
# this class is used for lists of nodes, lines, areas, parts, etc
class Item_List(list):
    def __init__(s):
        super().__init__() # these lists start empty

    def get_ids(s):
        # returns list of ids
        return [a.id for a in s]

    def get_next_id(s):
        # returns the next id to use
        ids = s.get_ids()
        minid = 0
        if len(ids) == 0:
            return minid    # list is empty so return the minid number
        else:
            ids = sorted(ids)
            maxid = ids[-1]
            unused = list(set(list(range(minid, maxid+2))) - set(ids))
            return unused[0]    # return first available id number
```

UNIQUELY IDENTIFY GEOMETRY, PG3

item.id = -1 by default

When an item is added to the list the next available id number is found

We set that id number in the item

Then add the item to the list

```
def append(s, item):
    # add item to the list and set it's id
    idnum = s.get_next_id()
    item.set_id(idnum)
    super().append(item)
    return item
```

PLOTTING SELECTED ELEMENT RESULTS, PG 1

When we plot results, we need to use interpolation to make the results look good.

The usual way to plot unstructured data in matplotlib is to make a regular grid using numpy and map our results onto it, using something like scipy griddata.

But that interpolates over large areas, and is not limited to the boundaries of our part and elements. Instead let's use pyplot: tripcolor or tricontourf

PLOTTING SELECTED ELEMENT RESULTS, PG 2

Each of these functions lets us plot unstructured data from a triangular mesh.

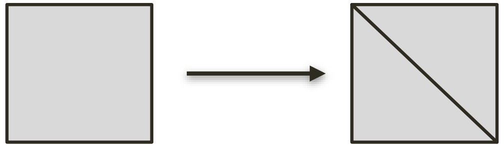
Tripcolor allows us to plot with continuous contours

Tricontourf allows us to plot filled contours

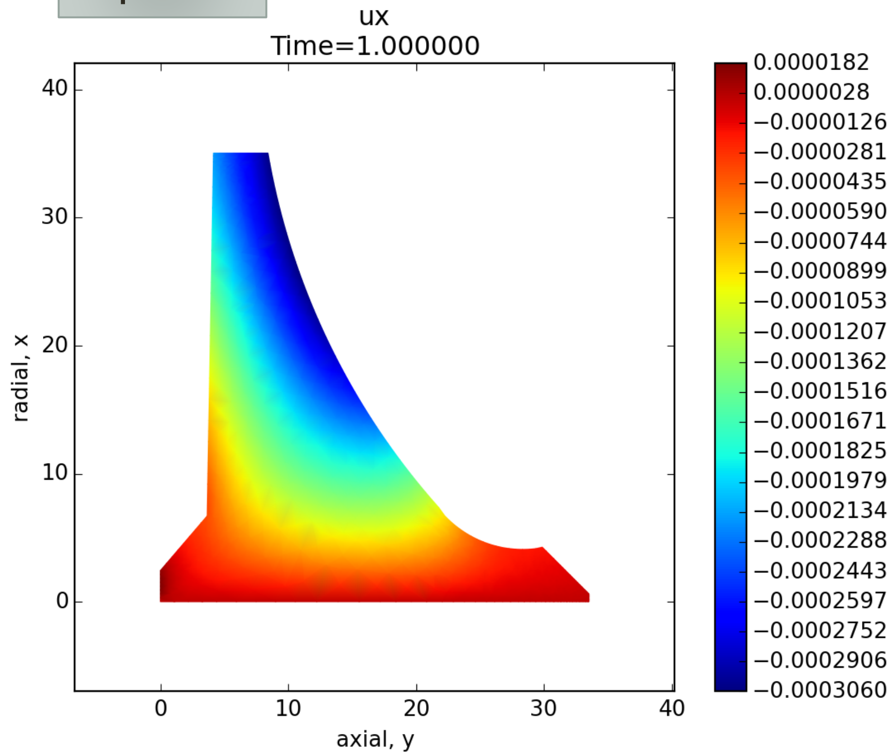
```
# plot using a gradient(shaded) or levels
if gradient:
    # This one is shaded
    plt.tripcolor(axials, radials, triangles, zs, shading='gouraud')
else:
    # this one is not shaded
    plt.tricontourf(axials, radials, triangles, zs, levels=tick_list)
```

PLOTTING SELECTED ELEMENT RESULTS, PG 2

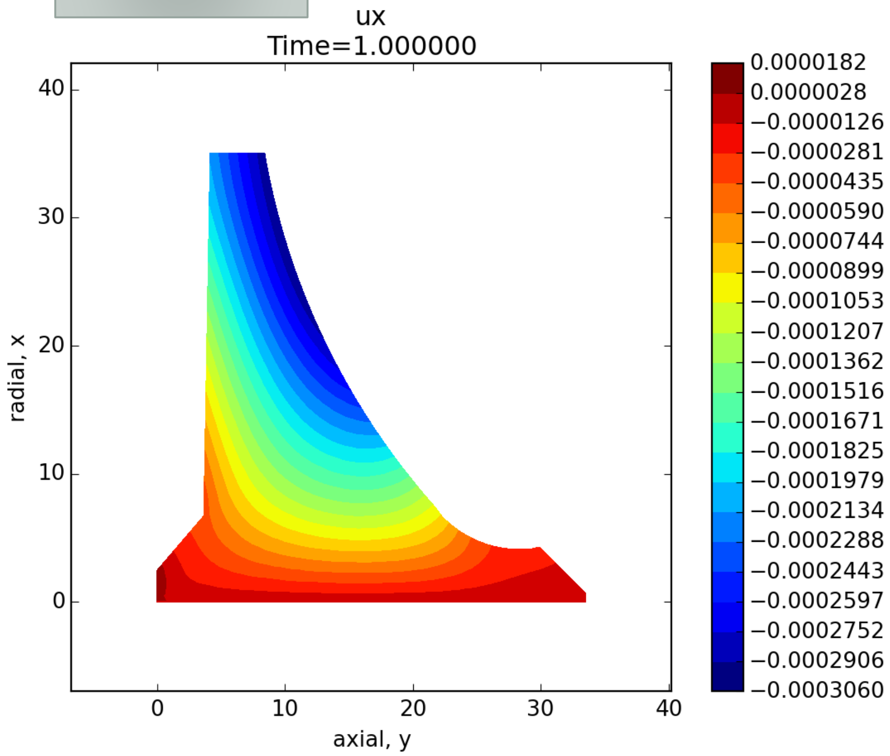
The only other trick we need to do is to have each element return its triangles. For a triangular element this is easy. For a quad element, we just have it split itself into two triangles.



tripcolor



tricontourf



IMPORT RESULTS, PG 1

The documentation listed results file formatting and variable names.

I wrote a function where I could pass in the format string, and the line

And get out a list of variables, correctly typed

This made data import very easy

```
# get the name to determine if stress or displ
line = f.readline()
fstr = "1X,I2,2X,8A1,2I5"
t = s.get_vals(fstr, line)
[KEY, NAME,NCOMPS,IRTYPE] = t
```

IMPORT RESULTS, PG 2

Results were very easy to process with my get_vals method

```
if mode == 'nodes':
    # node definition, store node numbers only
    t = s.get_vals(rfstr, line)
    [KEY, NODE, x, y, z] = t

elif mode == 'displ':
    # displacements
    t = s.get_vals(rfstr, line)
    [KEY, NODE, ux, uy, uz] = t

elif mode == 'stress':
    # stresses
    t = s.get_vals(rfstr, line)
    [KEY, NODE, sx, sy, sz, sxy, syz, szx] = t
```

DOWNLOAD PYCALCULIX ON GITHUB

https://github.com/spacether/pycalculix

Future distribution may be through PiPi or my website.

APPENDIX

IMPORT RESULTS, PG 2

```
def get_vals(s, fstr, line):
    # this returns a list of items based on an input format string
    res = []
    fstr = fstr.split(',')
    for item in fstr:
        if item[0] == "'":
            # strip off the char quaotes
            item = item[1:-1]
            # this is a string entry, grab the val out of the line
            ind = len(item)
            fwd = line[:ind]
            line = line[ind:]
            res.append(fwd)
        else:
            # format is: 1X, A66, 5E12.5, I12
            # 1X is number of spaces
            (m,c) = (1, None)
            m_pat = re.compile(r'^\d+') # find multiplier
            c_pat = re.compile(r'[XIEA]') # find character
            if m_pat.findall(item) != []:
                m = int(m_pat.findall(item)[0])
            c = c_pat.findall(item)[0]
            if c == 'X':
                # we are dealing with spaces, just reduce the line size
                line = line[m:]
            elif c == 'A':
                # character string only, add it to results
                fwd = line[:m].strip()
                line = line[m:]
                res.append(fwd)
```

This block is for quoted strings only

Add this portion of the line to the results

Format is mc

m = multiple

c = character

Add string to results

IMPORT RESULTS, PG 3

```
else:
    # IE, split line into m pieces
    w_pat = re.compile(r'[IE](\d+)') # find the num after char
    w = int(w_pat.findall(item)[0])
    for i in range(m):
        # only add items if we have enough line to look at
        if w <= len(line):
            substr = line[:w]
            line = line[w:]
            substr = substr.strip() # remove space padding
            if c == 'I':
                substr = int(substr)
            elif c == 'E':
                substr = float(substr)
            res.append(substr)
return res
```

Format is mcw
m = multiple
c = character is I or E
w = width

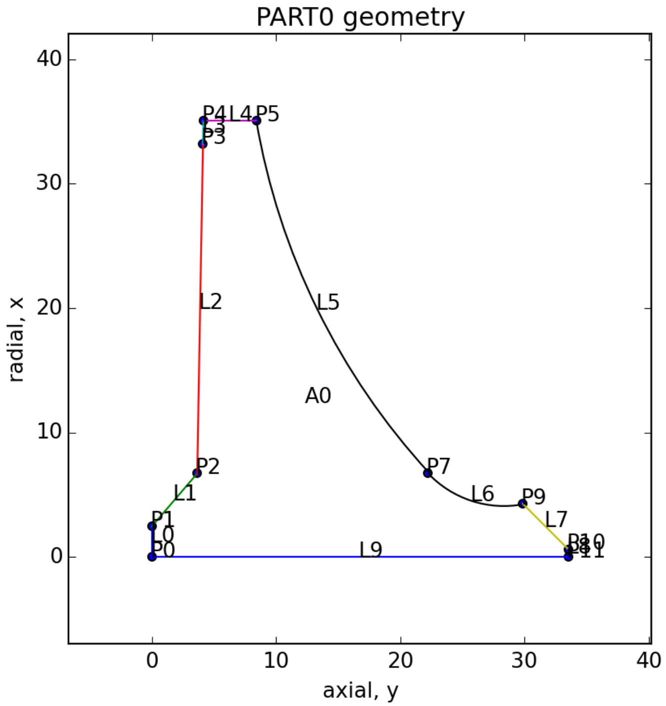
Cast as int or float

Add to results

File: dam.py

EXAMPLE: ANALYZING A DAM (BEETALOO DAM) PLANE STRAIN

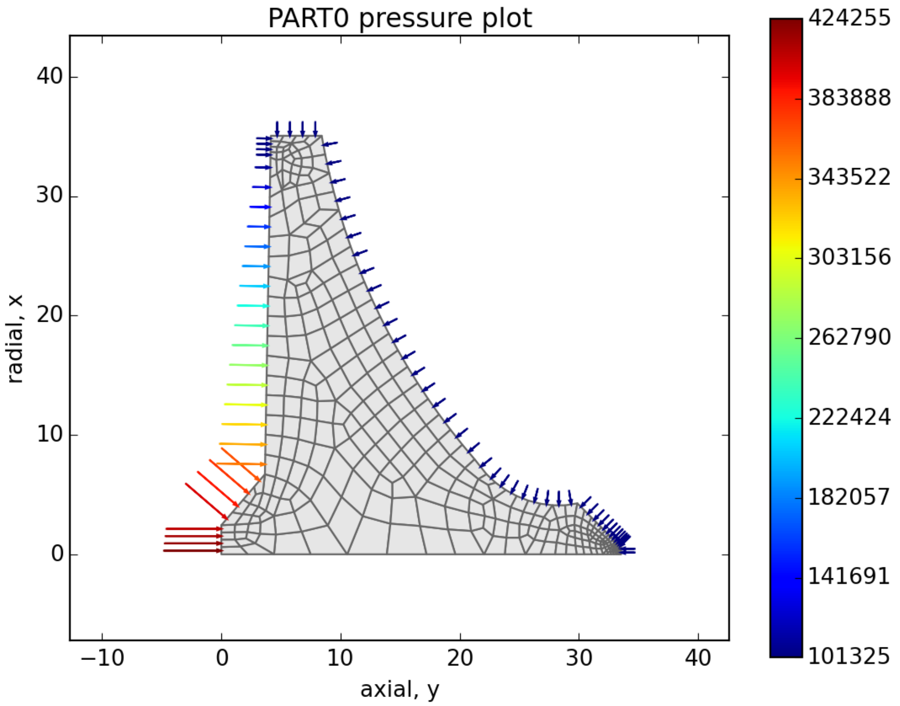
(meters)



Make part

Assign material

(Pa = N/(m^2))



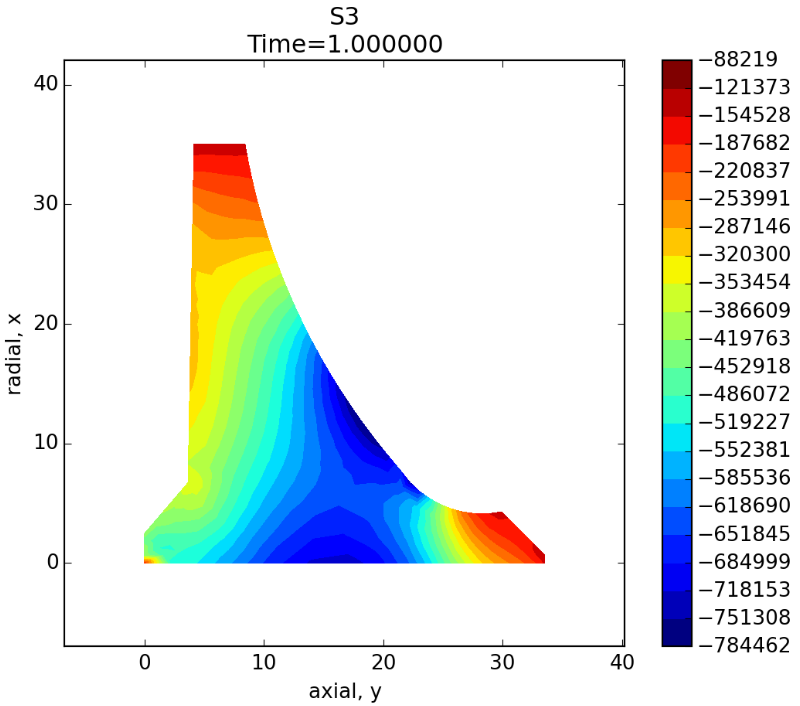
Mesh part

Apply constraints

Apply pressures

Apply gravity

(Pa = N/(m^2))



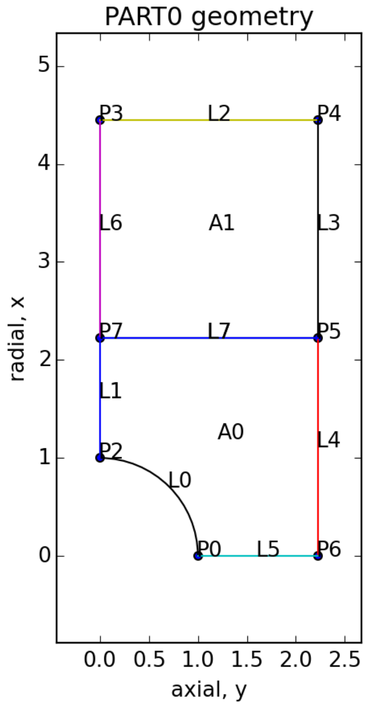
Solve model

View Results

File: hole_model.py

EXAMPLE: HOLE IN PLATE UNDER TENSION PLANE STRESS

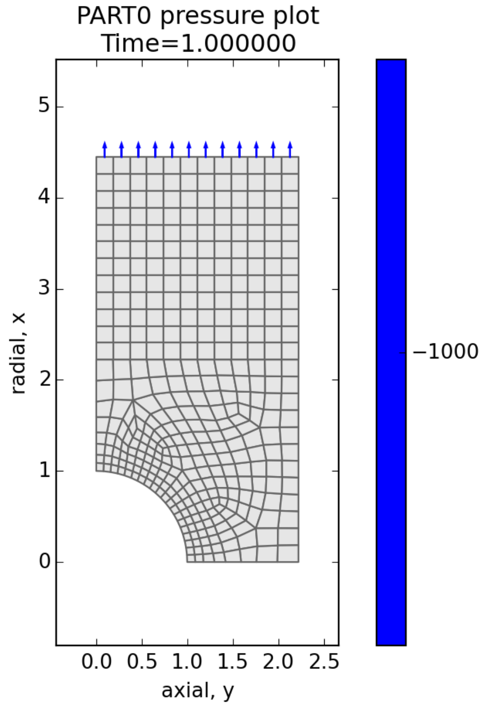
(meters)



Make part

Assign material

(Pa = N/(m^2))

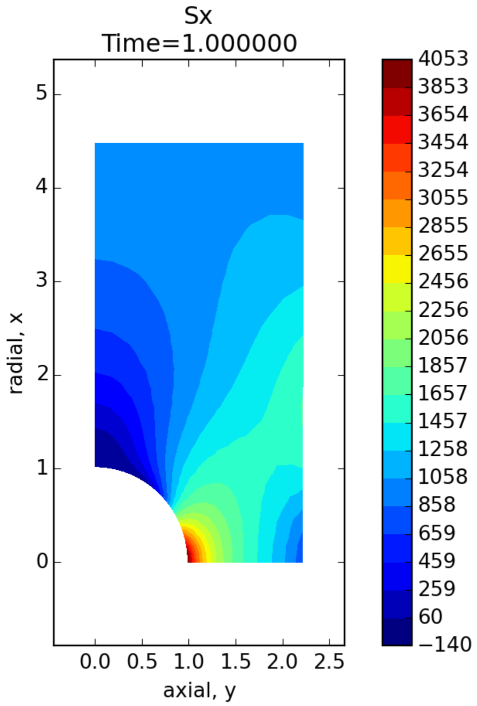


Mesh part

Apply constraints

Apply pressures

(Pa = N/(m^2))



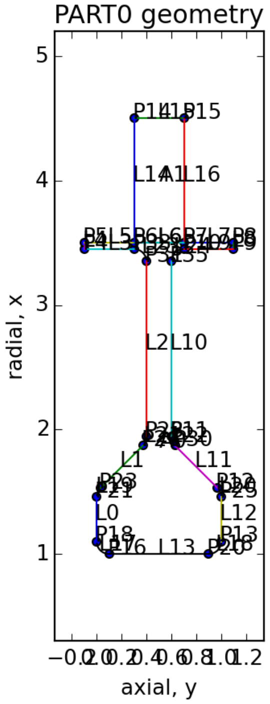
Solve model

View Results

File: compr_rotor_stage.py

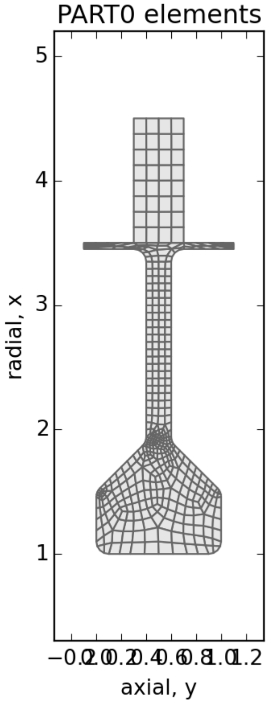
EXAMPLE: COMPRESSOR DISK OR TURBINE DISK AXISYMMETRIC

(meters)



Make part

Assign material

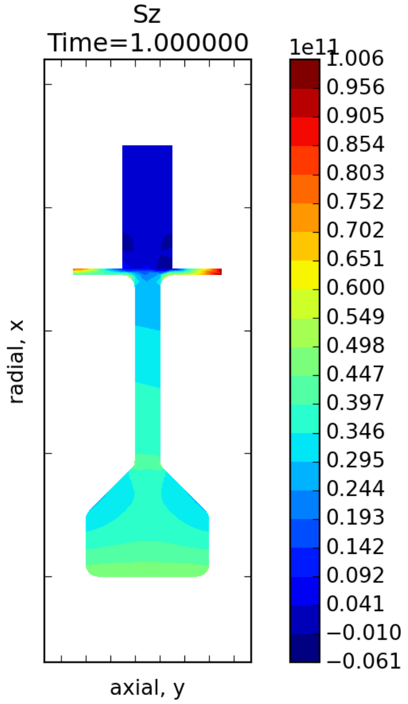


Mesh part, set thickness on airfoil

Apply constraints

Apply speed

(Pa = N/(m^2))



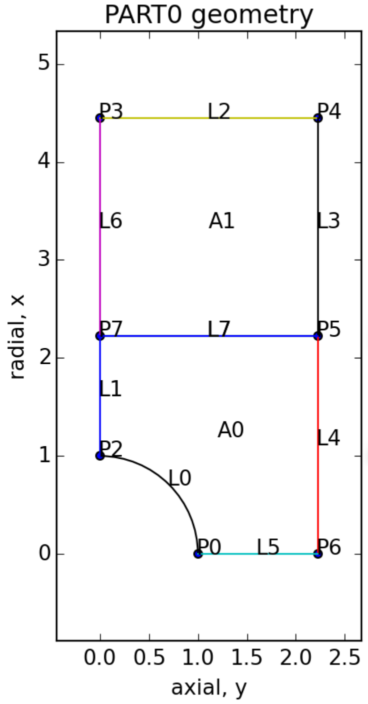
Solve model

View Results

File: hole_kt.py

EXAMPLE: DESIGN STUDY PETERSON TENSION HOLE IN PLATE, PG 1

(meters)



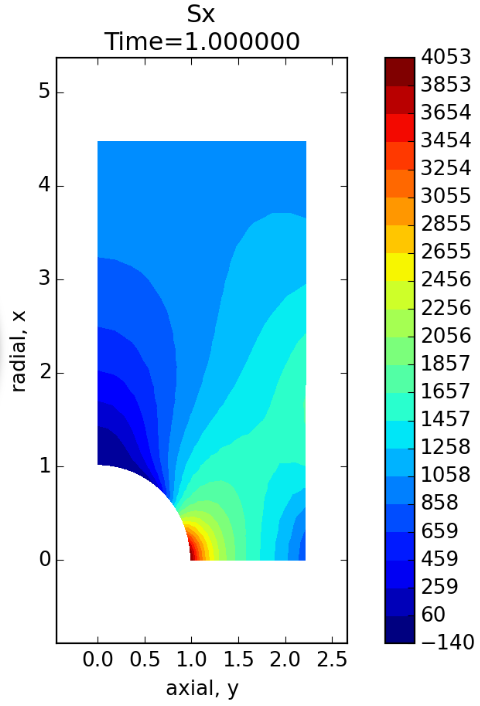
Make part

Assign material

Mesh Part

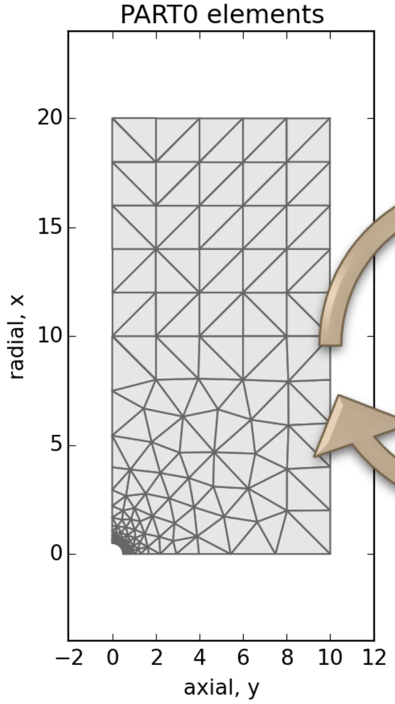
Apply constr + press

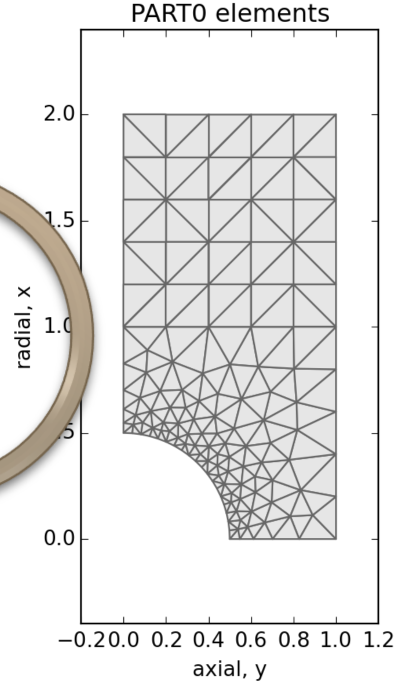
(Pa = N/(m^2))



Solve model

Extract Kt



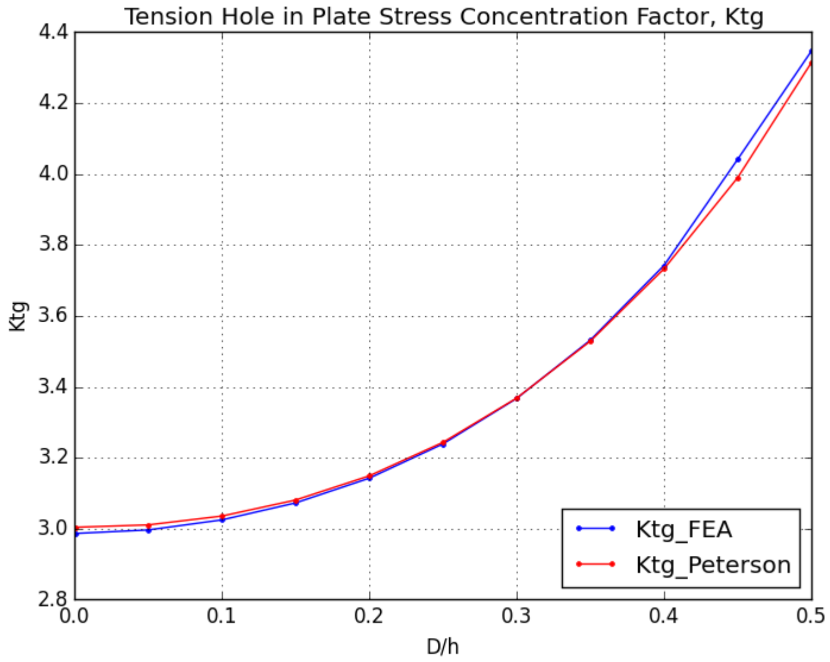


Run multiple models with a range of plate widths, using a constant hole size.

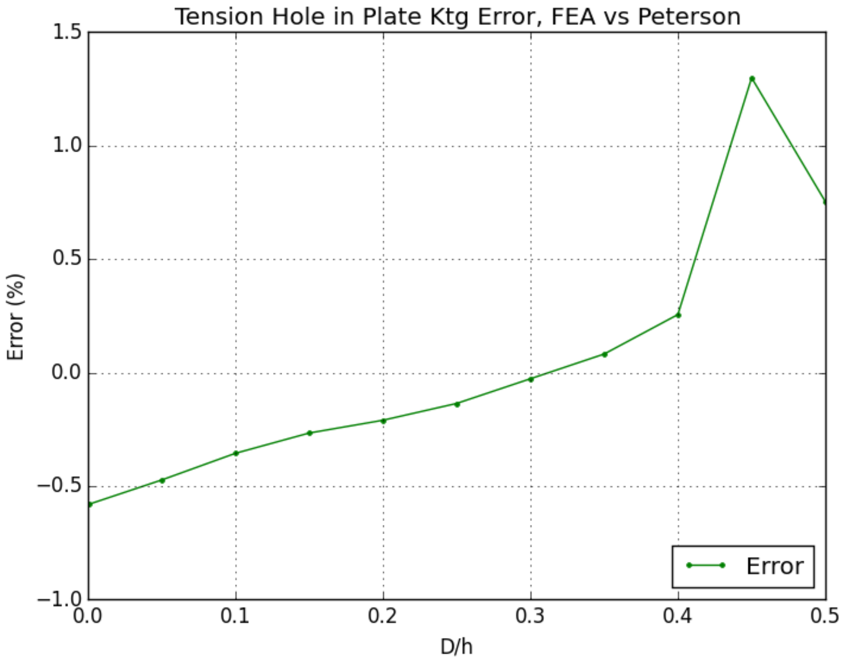
Compare Calculix FEA results with Peterson predicted results.

File: hole_kt.py

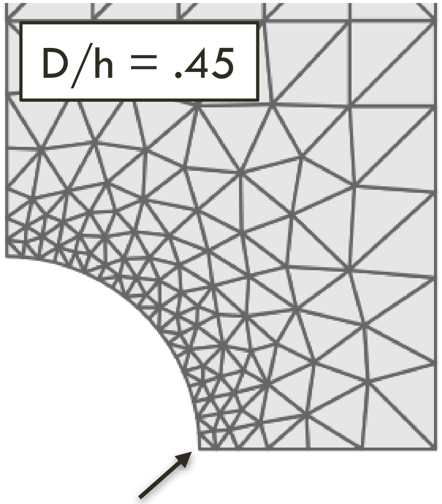
EXAMPLE: DESIGN STUDY
PETERSON TENSION HOLE IN PLATE, PG 2



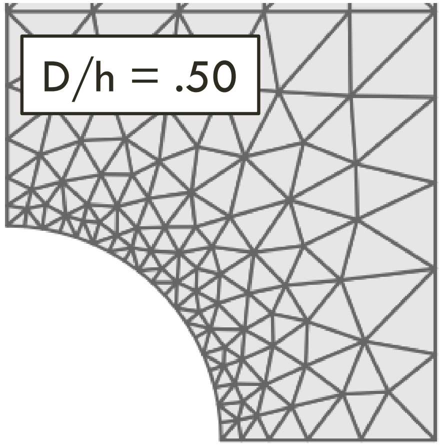
Run multiple models with a range of plate widths, using a constant hole size.
Compare Calculix FEA results with Peterson predicted results.



Calculix FEA results are accurate to within 1.5% of Peterson's results. Error jump is probably due to layout of local elements.
19 elements used on arc, 2nd order tris used



Error may be higher
Because only one element
on this corner
All other runs had 2 like below



File: hole_model.py

WALK THROUGH, HOLE IN PLATE, PG1

Import the pycalculix library and define a model

This model will hold all of our geometry, materials, loads, constraints, elements, and nodes.

```
from pycalculix import FeaModel

# Vertical hole in plate model, make model
proj_name = 'hole_model'
a = FeaModel(proj_name)
a.set_units('m')    # this sets dist units to meters, labels our consistent units
```

File: hole_model.py

WALK THROUGH, HOLE IN PLATE, PG2

Define the variables that we'll use to draw the part

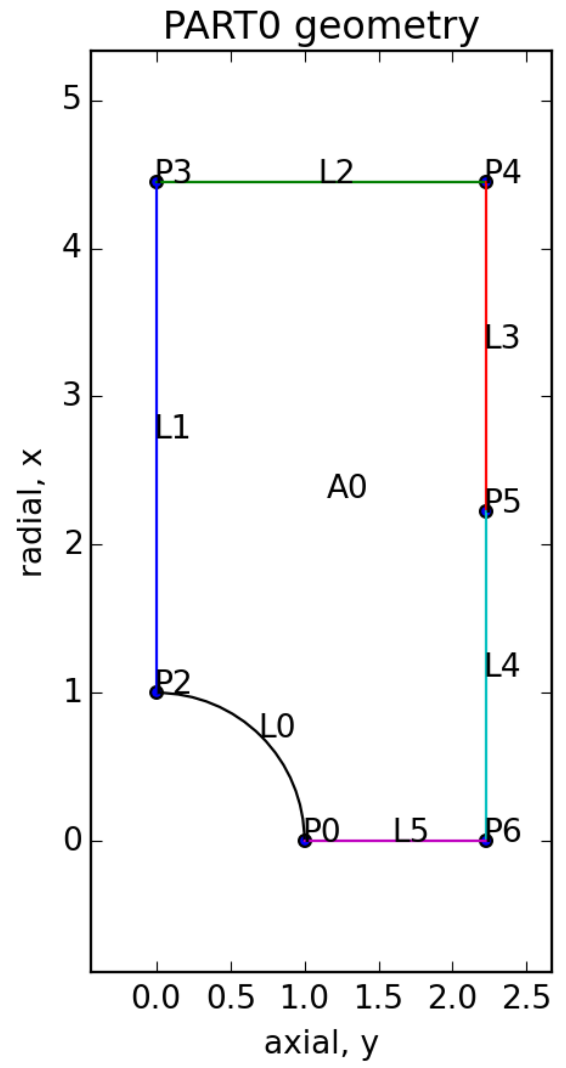
```python
# Define variables we'll use to draw part geometry
diam = 2.0 # hole diam
ratio = 0.45
width = diam/ratio   #plate width
print('D=%f, H=%f, D/H=%f' % (diam, width, diam/width))
length = 2*width  #plate length
rad = diam/2   #hole radius
vdist = (length - 2*rad)/2  #derived dimension
adist = width/2             #derived dimension
```

File: hole_model.py

WALK THROUGH, HOLE IN PLATE, PG3

Draw the part. We have to make a PartMaker instance to store the part.
Part must be drawn in CLOCKWISE direction
x = vertical axis, also known as the 'radial' axis
y = horizontal axis, also known as the 'axial' axis
Draw_line_rad = draw radial line (vertical)
Draw_line_ax = draw axial line (horizontal)

```
# Draw part geometry, you must draw the part CLOCKWISE
# coordinates are x, y = radial, axial
b = a.PartMaker()
b.goto(0.0,rad)
b.draw_arc(rad, 0.0, 0.0, 0.0)
b.draw_line_rad(vdist)
b.draw_line_ax(adist)
b.draw_line_rad(-length/4.0)
b.draw_line_rad(-length/4.0)
b.draw_line_ax(-(adist-rad))
b.plot_geometry(proj_name+'_prechunk') # view the geometry
```



File: hole_model.py

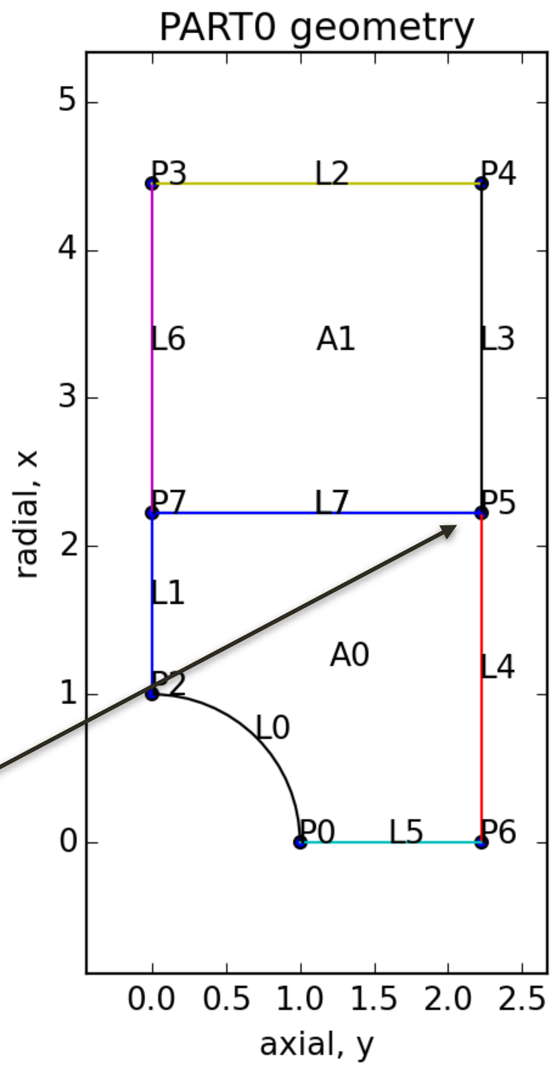
WALK THROUGH, HOLE IN PLATE, PG4

Chunking tells the program to try to cut the area into smaller pieces

It cuts the part at points. It draws a perpendicular line then cuts the part with it.

Chunking can help you make a better quality mesh.
It is required for CGX meshing, but not for GMSH meshing.

```
# Cut the part into easier to mesh areas
b.chunk() # cut the part into area pieces so CGX can mesh it
b.plot_geometry(proj_name+'_chunked') # view the geometry
```



Area was chunked at P5

File: hole_model.py

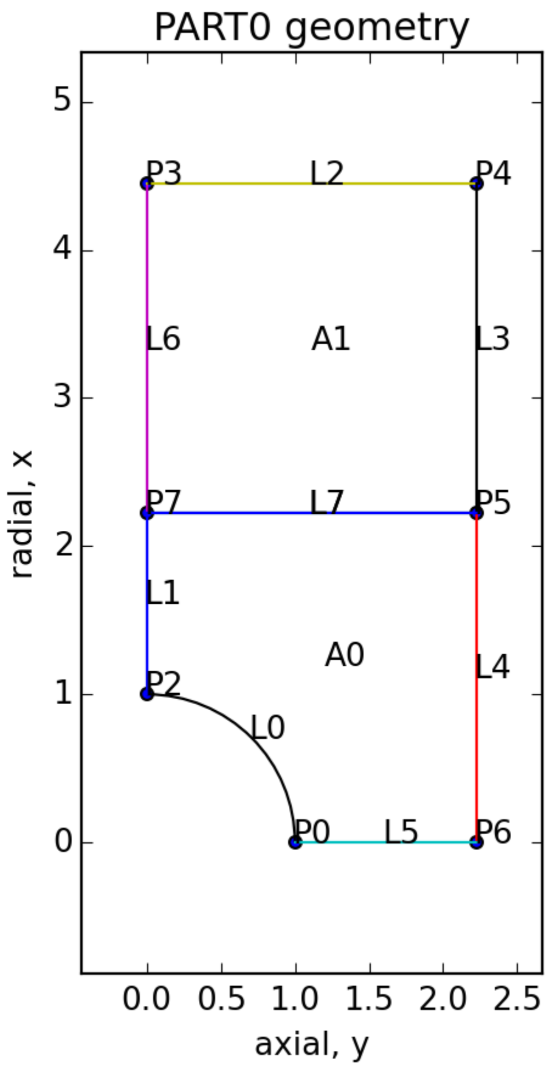
WALK THROUGH, HOLE IN PLATE, PG5

Sets the loads and constraints

Positive pressures push on the part. Negative pressures pull on the part.

Note: we can do this either before or after meshing because the program stores loads on geometry (points, lines, areas) rather than the mesh.

```
# set loads and constraints
a.set_load('press',b.top,-1000)
a.set_constr('fix',b.left,'y')
a.set_constr('fix',b.bottom,'x')
```

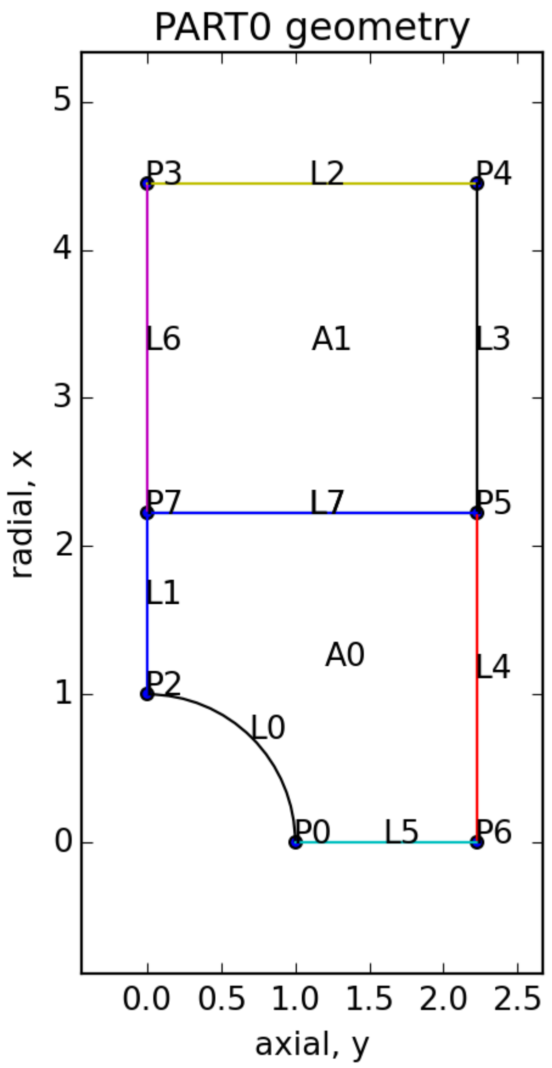


File: hole_model.py

WALK THROUGH, HOLE IN PLATE, PG6

Set the part material

```
# set part material
mat = a.MatlMaker('steel')
mat.set_mech_props(7800, 210000, 0.3)
a.set_matl(mat, b)
```



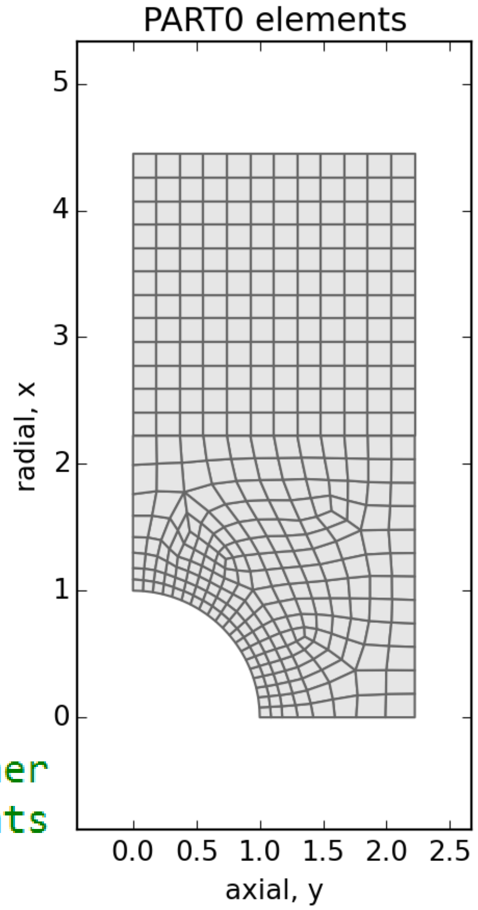
File: hole_model.py

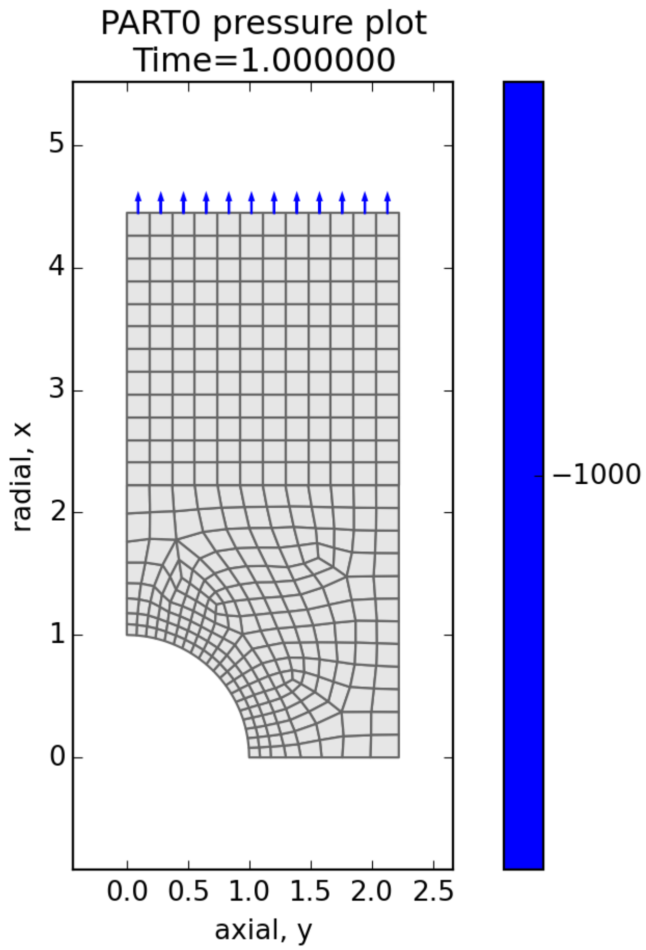
WALK THROUGH, HOLE IN PLATE, PG7

Mesh the part

set_eshape(shape=‘quad’ or ‘tri’, order=1 or 2)
set_etype(part, etype, thickness)
etype:
- ‘plstress’ = plane stress, thickness is required
- ‘plstrain’ = plane strain, thickness is required
- ‘axisym’ = axisymmetric, thickness is not required

```
# set the element type and mesh database
a.set_eshape('quad', 2)
a.set_etype(b, 'plstress', 0.1)
b.get_item('L0').set_ediv(20) # set element divisions
a.mesh(1.0, 'gmsh') # mesh 1.0 fineness, smaller is finer
b.plot_elements(proj_name+'_elem')   # plot part elements
b.plot_pressures(proj_name+'_press')
```





File: hole_model.py

WALK THROUGH, HOLE IN PLATE, PG8

Make and solve the model.

Python console output on the right.

```python
# make and solve the model
mod = a.ModelMaker(b, 'struct')
mod.solve()
```

```
Solving done!
Reading results file: hole_model.frd
Reading nodes
Reading displ storing: ux,uy,uz,utot
Reading stress storing: Sx,Sy,Sz,Sxy,Syz,Szx,Seqv,S1,S2,S3
Reading strain storing: ex,ey,ez,exy,eyz,ezx,eeqv
Reading force storing: fx,fy,fz
The following times have been read:  [1.0]
Done reading file: hole_model.frd
Results file time set to: 1.000000
```

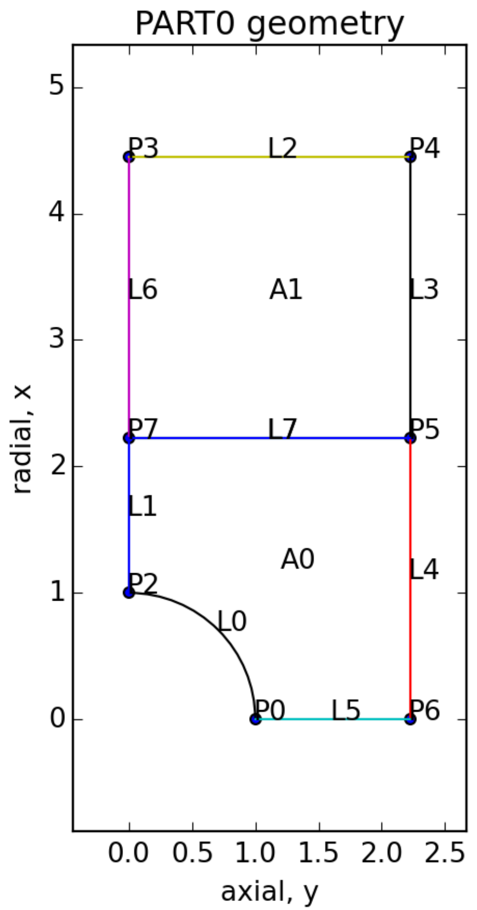
File: hole_model.py

WALK THROUGH, HOLE IN PLATE, PG9

Query our results. Check the max stress and the reaction forces.

Python console output below

```
# view and query results
sx = mod.rfile.get_nmax('Sx')
print('Sx_max: %f' % (sx))
[fx, fy, fz] = mod.rfile.get_fsum(b.get_item('L5'))
print('Reaction forces (fx,fy,fz) = (%12.10f, %12.10f, %12.10f)' % (fx,
```




```
Sx_max: 4052.960000
Reaction forces (fx,fy,fz) = (0.0000000000, 0.0000000000, 0.0000000000)
```

The reaction force output shouldn’t be zero here.
These results come directly from the results FRD file.
This is a bug which needs to be fixed in the solver.

File: hole_model.py

WALK THROUGH, HOLE IN PLATE, PG10

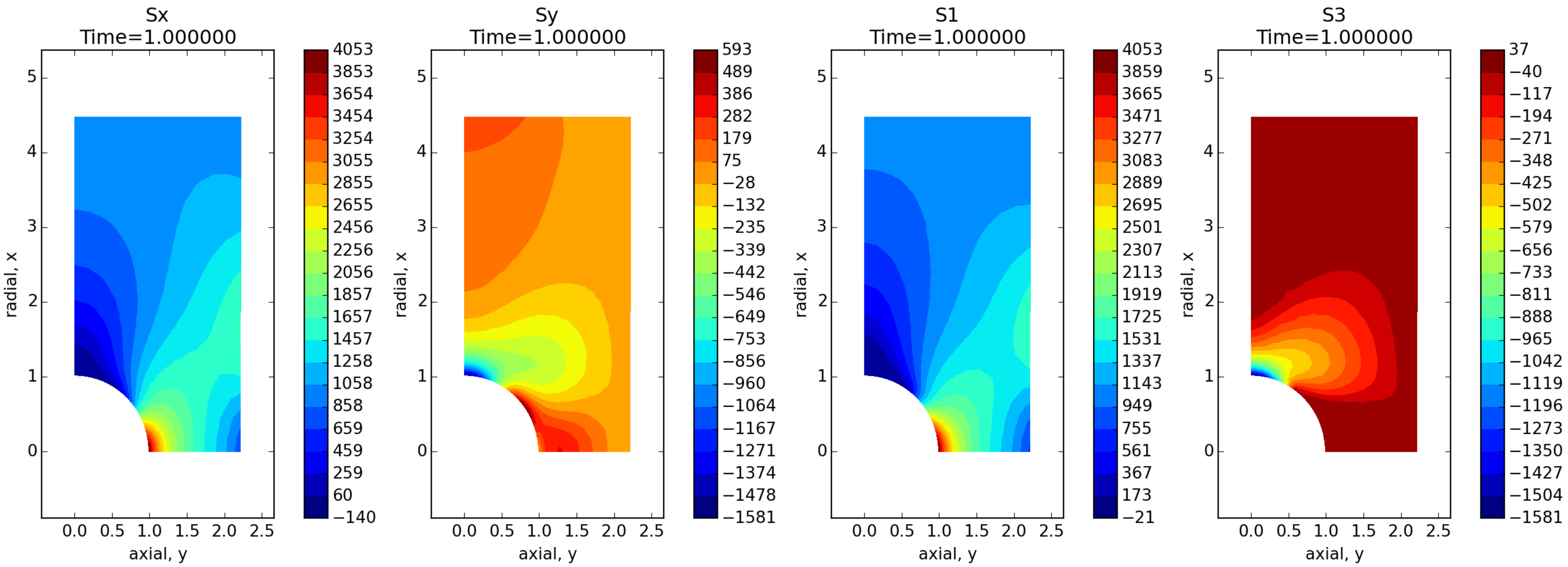
Plot our results.

Interactive plotting is suppressed with the display variable, but files are saved.

```
# Plot results
disp = False
mod.rfile.lplot('Sx', proj_name+'_Sx', display=disp)
mod.rfile.lplot('Sy', proj_name+'_Sy', display=disp)
mod.rfile.lplot('S1', proj_name+'_S1', display=disp)
mod.rfile.lplot('S2', proj_name+'_S2', display=disp)
mod.rfile.lplot('S3', proj_name+'_S3', display=disp)
mod.rfile.lplot('Seqv', proj_name+'_Seqv', display=disp)
mod.rfile.lplot('ux', proj_name+'_ux', display=disp)
mod.rfile.lplot('uy', proj_name+'_uy', display=disp)
mod.rfile.lplot('utot', proj_name+'_utot', display=disp)
```

File: hole_model.py

WALK THROUGH, HOLE IN PLATE, PG11, PLOTS



File: hole_model.py

WALK THROUGH, HOLE IN PLATE, PG12, PLOTS